Induction and Co-Induction in Sparkle

Leonard Lensink[1], Marko van Eekelen[2]

[1]Email:llensink@xs4all.nl
[2]Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, Toernooiveld 1, Nijmegen,
6525 ED, The Netherlands; Phone +31 (0)24-3653410;
Email: M.vanEekelen@niii.ru.nl

January 19, 2005

Abstract: Sparkle is a proof assistant designed for the lazy evaluating functional programming language Clean. It is designed with the clear objective in mind that proofs of first order logical predicates on programs should be as easy as possible for programmers. Since recursion is at the heart of functional languages, support for recursive reasoning should be as exhaustive as possible. Until now the induction tactic in Sparkle was rather basic. We have extended Sparkle with two standard techniques to perform inductive and co-inductive reasoning about programs. By means of examples, we show from research papers how they benefit the programmer. More importantly, we propose a new technique to derive induction schemes for mutually recursive programs by using strongly connected components of complete call graphs to derive a well founded ordering and induction scheme. These induction schemes can be used to semi-automatically prove properties of programs out of reach of other automated proof techniques. Together this extends the realm of programs for which easy proofs in Sparkle can be constructed by the programmer.

1 Introduction

In this introduction we will first briefly acquaint the reader with the proof assistant Sparkle, before the subject of induction and co-induction within Sparkle is introduced.

1.1 A brief introduction to Sparkle

Sparkle is the integrated proof assistant available with the lazy functional programming language Clean [MvE01]. The main purpose of the theorem prover is to help the programmer prove properties of programs. Several features are especially useful.

Firstly, the reasoning takes place in first order logic on the level of the program itself, so no translation of the program is needed. Secondly, the theorem prover is partly automated. And finally, the theorem prover is integrated into the development environment of Clean.

All these features are intended to encourage people who are not well versed in proof assistants to make use of the advantages a correctness proof can give. A brief description of Sparkle's capabilities can be found in [MdM01].

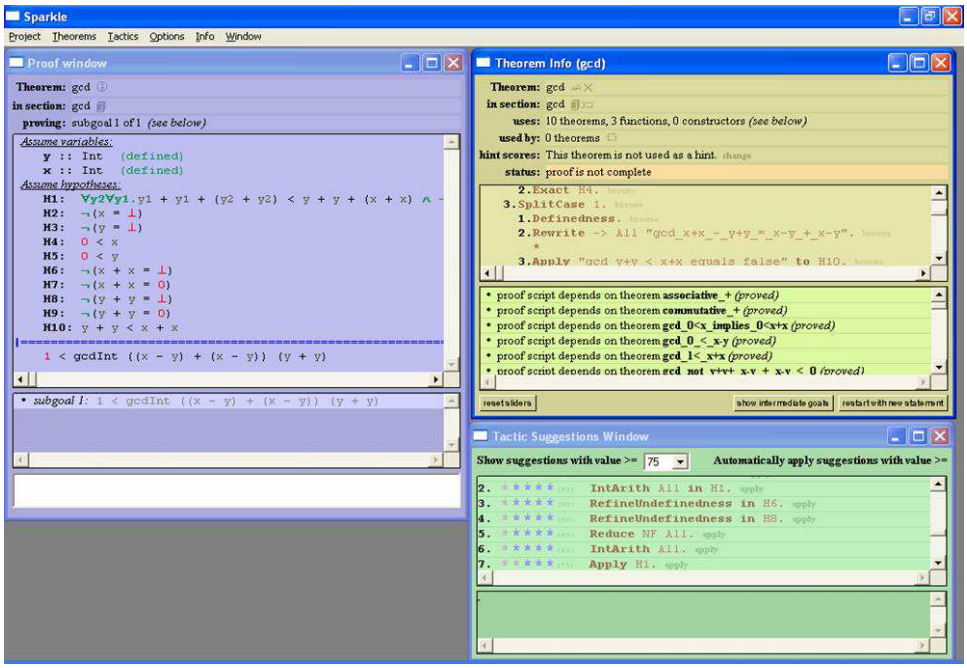

Figure 1: Sparkle in Action

Reasoning in sparkle is similar to other theorem provers. The user starts off with a goal, a property, that needs to be proven. A goal has a goal environment in which the local hypotheses and introduced variables are

stored. Several tactics are available to the user, which can be applied to the goal. Each tactic is a function from a goal and its environment to a list of goals and environments. Whenever all the goals are proven, the property is proven correct.

In order to make life as easy as possible for the programmer, Sparkle also contains a hint mechanism. This means it will try to guess which tactic might be applicable at a certain time. These hints are assigned a score. The higher the score, the more likely the tactic will be useful to the programmer in order to complete the proof. By having Sparkle apply the highest scoring tactics automatically, propositions can be proven without any user intervention whatsoever.

1.2 Using Sparkle for Haskell programs

Although all proofs and tactics are done within the Clean environment they are not limited to this environment. The tactics can be implemented, and some already are, in other theorem provers. Sparkle can also be used to prove properties of Haskell programs. A fully automated Haskell to Clean converter is available, called Hacle[1]. The similarities of both functional languages make reasoning on the source code level of Haskell programs possible.

To show how all the programs and proofs are constructed in Sparkle, all the programs and sections containing the proof are available to the interested reader. The extended Sparkle version, as well as the Clean compiler needed to compile the programs are at the same site[2].

1.3 Induction and co-induction in Sparkle

Many different proof techniques have been incorporated into Sparkle, but so far only a basic form of induction was available to the user. This tactic generated an induction scheme for directly recursive algebraic data types and integer functions.

In order to improve the usefulness of the theorem prover we have implemented multi-predicate induction over mutually recursive algebraic data types. Since this kind of induction has been implemented a few times already, we briefly introduce this tactic by means of an example from the literature in section 2.1.

Besides an extension of the induction proof principle, co-inductive proof techniques by means of bisimulation are introduced. Since Clean is a lazy

[1] Hacle can be found here: www-users.cs.york.ac.uk/ mfn/hacle/

[2] http://www.xs4all.nl/ llensink/Sparkle.zip

functional language, many properties of lazily defined data types and functions can only be proven using this tactic. In section 2.2 we will demonstrate by means of an example taken from the literature how this new bisimulation tactic can be used for proofs of interesting properties of non terminating functional programs.

Another new feature is the possibility to have the proof assistant generate schemes for predicates involving mutually recursive function calls. This method constructs induction schemes based on the way the mutually recursive function definitions are interdependent. New in this induction method, compared to already existing methods is that the well-foundedness ordering that is the basis of the induction scheme is derived from the arguments of the recursive function calls in the function definitions. A graph is constructed and analyzed to find the best combination of arguments to build an ordering from. Although, there are more sophisticated methods of using call graphs to prove termination of recursive functions, we argue that our method, despite its simplicity yields good results and is especially suitable for Sparkle, because it generates an induction scheme that can be used in combination with other tactics to prove properties of a wide range of recursive programs. A detailed explanation of the implementation as well as a comparison with one of the other methods can be found in section 3.

2 Standard techniques for recursive functions

In this section we will show by example how the implementation of two standard techniques of constructing inductive and co-inductive proofs is realized in Sparkle. The examples, taken from research papers, show how they enable the programmer to reason about her code with ease.

2.1 Induction on mutually recursive data types

Mutual recursion in algebraic data types require proof techniques that can handle the inter-dependency between the constituent parts. A well known method, employed for instance in the HOL theorem prover [MJCG93], uses multi-predicate induction. In this section we will briefly describe our implementation of it by means of an example.

One of the properties of Clean and Haskell is that all the algebraic data type definitions must be guarded with a constructor and each constructor must be unique. This means that we do not have to worry about non-productive data types or ambiguous derivations. The only restriction to the (mutually) recursive algebraic data type is that at least one of the recursive

definitions has a non-recursive part. If not, we don't have a smallest element to build our ordering on.

Another point of interest is that an expression of a certain data type may not terminate or be undefined. In Sparkle this is represented by $\perp$. Since predicates on non terminating functions may not always be allowed they must be deemed admissible. The way admissibility is determined within Sparkle is based on Paulson's criterium as stated in his Logic and Computation book [Pau87]

Take for instance a simple expression language consisting of arithmetic and boolean expressions. This example can be found on the HOL website[1]. Arithmetic expressions can contain a boolean condition, while boolean expressions may be a comparison between arithmetic expressions.

```
::Aexp a = Var a
         | Num Int
         | Sum (Aexp a) (Aexp a)
         | If (Bexp a) (Aexp a) (Aexp a)
::Bexp a = Less (Aexp a) (Aexp a)
         | And (Bexp a) (Bexp a)
```

On this small language substitution and evaluation functions are defined. The substitution function performs a substitution, defined as a map from pointers to arithmetic expressions, on an expression. The evaluation function reduces an expression to a value within a certain environment. That environment is defined as a map from pointers to integer values. To conserve space we only give the function body of the evala function as an example.

```
evala :: (a->int) (Aexp a) -> int
evala e (If b a1 a2)
   | evalb e b = evala e a1
   | otherwise = evala e a2
evala e (Sum a1 a2) = (evala e a1) + (evala e a2)
evala e (Var v) = e v
evala e (Num n) = n

evalb :: (a->int) (Bexp a) -> bool
substa :: (a -> Aexp a) (Aexp a) -> Aexp a
substb :: (a -> Aexp a) (Bexp a) -> Bexp a
```

We want to prove that it does not matter whether we evaluate all the substitutions and use that as an environment or we perform all the substitutions first and then evaluate the resulting expression. Formulated within Sparkle it gives us the following property to prove.

[1] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/HOL/Induct/ABexp.html

Property 2.1 $\forall$`env` s a.`evala env` (`substa` s a) = `evala` (λx.`evala env` (s x)) $a \wedge \forall$`env` s b.`evalb env` (`substb` s b) = `evalb` (λx.`evala env` (s x)) b

Sparkle's new induction tactic will recognize that the induction will be done on the variables a and b, of a mutual recursive type and offers an induction scheme whereby for each non recursive part of the data types a proof of the property is required, while for each recursive definition an antecedent is constructed. Sparkle will generate eight subgoals that need to be proven. Two of them are the cases when a or b are $\perp$. To show all the goals will take up too much space, but from goal 2.2. the structure of the other goals will be readily apparent.

Goal 2.2 $\forall b1\ a1\ a2$.
($\forall$`env` s.`evalb env` (`substb` s $b1$) = `evalb` (λx.`evala env` (s x)) $b1$) $\wedge$
$\forall$`env` s.`evala env` (`substa` s $a1$) = `evala` (λx.`evala env` (s x)) $a1$) $\wedge$
$\forall$`env` s.`evala env` (`substa` s $a2$) = `evala` (λx.`evala env` (s x)) $a2$)) $\rightarrow$
$\forall$`env` s.`evala env` (`substa` s (`IF` $b1$ $a1$ $a2$)) =
`evala` (λx.`evala env` (s x)) (`IF` $b1$ $a1$ $a2$)

This proposition is proven in Sparkle by assuming the induction hypotheses and then reducing the evala function.

In this section we have shown how Sparkle derives an induction scheme for mutually recursive data types. The derived subgoals are subsequently easily proven by reducing the evaluation function. There is almost no interaction required from the programmer.

2.2 Co-Induction

Co-recursive programs are less well known than recursive programs, but the growing insight that they can be a very useful and elegant way of programming, means that proof techniques and tactics need to be made available to programmers.

The power of co-recursion is shown in an McIlroy's article on power series [McI99]. This article shows how power series can be implemented using streams of fractions. These infinite streams of integers represent the coefficients of the power series.

For instance the power series for cos x; $1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$. is represented by the infinite list of fractions: [1,0,-1/2,0,1/24,0,-1/720,...]. Using infinitely recursing functions on these streams mathematical operations like addition, subtraction and even calculus can be defined on these power series.

An example of how simple these definitions are is the implementation of the sinus and co-sinus function:

```
sin x = integral cos x
cos x = 1 - (integral sin x)
integral x = [0:int1 x 1] where
  int1 [x:xs] n = [x/n:int1 xs (n+1)]
```

Since these programs operate on an infinite list, they will never terminate. This makes it impossible to use induction to conduct our proofs. A co-inductive proof tactic is needed. Several methods for dealing with co-recursive programs are described in [JG04]. Fixed point induction, fusion, bisimulation and the approximation lemma are mentioned. Our tactic uses the bisimulation relationship suggested by the property one is trying to prove. As described in [BJ96], a bisimulation is a relation on a set of (infinite) elements of a certain data type with a co-algebraic structure defined by its destructor functions. For lists this relationship would be: $R(a, b) \rightarrow \texttt{hd}\ a = \texttt{hd}\ b \wedge R(\texttt{ta}\ a, \texttt{tl}\ b)$. This relationship defines the co-inductive proof principle;

Theorem 2.3 $\forall a, b.R(a, b) \rightarrow \forall a, b.a = b$

Suppose we want to prove the mathematical property of the sinus function that $-(\sin x) = \sin(-x)$, using the co-inductive proof principle. Applying the bisimulation tactic directly on this equation will not yield an easy proof. There is another recursive function within the definition of sin. As a general rule it is more productive to proof properties of that inner recursion first. So choose another property to help us:

Property 2.4 $\forall z\ n. - (\texttt{int1}\ z\ n) = \texttt{int1}\ (-z)\ n$

Applying the bisimulation tactic to this equality relationship we get two proof obligations:

1. First we have to prove that the heads of both infinite streams are the same: `hd -(int1 z n) = hd (int1 (-z) n)`. By reducing int1 in both sides of the equation we get: `[-x/n:-int1 xs (n+1)] = [-x/n:int1 -xs (n+1)]`. Clearly the `hd` of those two lists is the same.

2. Subsequently we have to prove that the `tl` of both infinite streams satisfy the stated equality relationship. Choose $z =$ `xs` and $n = n + 1$ to see that it satisfies the required relationship.

Once we have proven property 2.4, it is easy to prove that -(sin x) = sin (-x).

In this section we have shown how proofs on recursive functions that do not terminate can be conducted within Sparkle. As demonstrated by this example, it is fairly easy for a programmer to prove interesting properties using the bisimulation tactic.

3 Induction on mutually recursive functions

The multi-predicate induction principle for algebraic data types, although useful in many instances, will not make easy proofs possible for a large group of programs that need more sophisticated induction schemes.

For instance this function to calculate the greatest common divisor is not easily within reach of conventional induction. The way the function is structured, with either the first or the second argument decreasing by an arbitrary amount, makes it hard to find a proper induction scheme. For reasons of presentation we assume that integers can be represented by natural numbers, disregarding any overflow issues.

```
gcd :: Int Int -> Int
gcd 0 y = y
gcd x 0 = x
gcd x y
  | x < 0 || y < 0 = abort ("Arguments must be positive")
  | y < x          = gcd (x-y) y
  | otherwise      = gcd  x   (y-x)
```

A different kind of induction scheme is needed for functions where the arguments are combined or switched around. Functions where the recursive structure of its definition does not match the structure of the data type of its arguments need a different kind of scheme as well. This new kind of induction scheme that matches the structure of the function definition will make easy proofs possible.

We will propose a method that makes use of call graphs to determine which arguments should be used to construct a well founded ordering. This ordering will be the basis of the induction scheme.

In the next section we first describe the tactic we created to derive induction schemes for (mutually) recursive functions. Then we briefly describe what kind of related work is done on the subject of terminating recursive functions and the derivation of induction schemes. Finally we compare our method with the size change principle. One of the methods from the related work section.

3.1 Tactic for deriving induction schemes for mutually recursive functions

When trying to find a well founded ordering for mutually recursive functions one quickly runs into the problem that the number and position of arguments for each function can be different. Furthermore, applications of other functions not included in the mutually recursive function set make it hard to track the arguments that should be used for the well founded ordering. To track the way the original arguments are used in subsequent recursive function calls a graph is built. Each argument of the mutually recursive functions is a vertex and the way arguments are interdependent is represented by directed edges.

3.1.1 Creating a call graph

Suppose there is a set of mutually recursive functions $S = \{f_1, .., f_n\}$ Let f_k be one of those mutually recursive functions. Graph G is a pair (E, V) of edges E and vertices V.

For each function definition, for instance f_k the arguments $a_1, .., a_j$ of the function definition are added to the set of vertices. For all function calls $f_g \in S$ in the body of the function definition, the variables that are used in the f_g function call are added to the set of vertices V and edges from $(f_k\ a_i)$ to $(f_g\ a_i)$ are added to the set of edges E.

For instance, take these function definitions:

```
f a1 a2 =
    | a1 < a2   = f (a1 + a2) a2
    | a1 == a2  = a2
    = g a1 a2
g a1 a2 = f a2 a1
```

They yield the directed graph $G = (V, E)$, where $V = \{(\texttt{f a1}), (\texttt{f a2}), (\texttt{g a1}), (\texttt{g a2})\}$ and $E = \{((\texttt{f a1}), (\texttt{f a1})), ((\texttt{f a1}), (\texttt{f a2})), ((\texttt{f a2}), (\texttt{f a2})), ((\texttt{f a1}), (\texttt{g a1})), ((\texttt{f a2}), (\texttt{g a1})), ((\texttt{g a1}), (\texttt{f a2})), ((\texttt{g a2}), (\texttt{f a1}))\}$.

Note that the first occurrence of a recursive function determines towards which recursive call the arguments are counted. A recursive call like `f a1 = g (f a1)` will yield the (sub)-graph: $V = \{(\texttt{f a1}), (\texttt{g a1})\}$, $E = \{((\texttt{f a1}), (\texttt{g a1}))\}$.

The only arguments that can possibly be of value to an induction argument are the ones that are able to create an infinite chain. Otherwise a simple exhaustion argument will do. Since there is a finite number of arguments and functions, only the arguments on a cycle in the graph are

candidates for a well founded relation. Elements on a graph are on a cycle if their vertices are in the same strongly connected component. Using Tarjan's [Tar72] algorithm we can find these components quickly.

3.1.2 Building a well founded relationship

Take for instance these mutually recursive functions, where arguments are combined and switched around.

```
f a b c = g (c+b) a
g x y   = f x y 1
```

The graph representing this function shows that f a b and g x y are strongly connected. The ordering we impose on this set of mutually recursive functions therefore has to take into account that for each function call f inside the function body of g, the arguments x and y combined have to be smaller than a and b together.

The key idea of this paper is how we construct a well founded relationship from these strongly connected components:

- For each possible transition between recursive functions $f_i \rightarrow f_j$, all vertices a_i and a_j that are in the same strongly connected component are added. If they are not, the lexicographical ordering of them is constructed.
- If a_i is an algebraic data type, a size function that counts the number of constructors in the definition is created.

In order to prove that the resulting function signifies a well founded relationship we have to prove that the functions we have constructed satisfy the definition of well foundedness:

Definition 3.1 $WF(R) \equiv \forall P.(\exists w.P(w)) \rightarrow \exists min.P(min) \wedge \forall b.R\ b\ min \rightarrow \neg P(b)$

It formalizes the notion that for any non empty subset of elements of a set, there is a smallest element in that subset. This means that there can not be an infinitely descending chain of smaller elements.

- This means that for our addition and size functions we have to prove that if we have a well founded relationship $<$ for natural numbers or strings, the relationship constructed by mapping the arguments onto integers is well founded as well.

Proposition 3.2 $\forall f.\forall P.(\exists w.P(f\ w)) \rightarrow \exists min.P(min) \wedge \forall b.f\ b < f\ min \rightarrow \neg P(f\ b)$

Proof 3.3 *Suppose we have a non-empty well founded set S, a subset of the set of natural numbers or strings. We define T as* $\{x|b \in S \rightarrow x = f\ b\}$. *Since S is non empty, and f is a total function, T is non empty as well. Both addition and size functions are total. Since S is well founded, we have a smallest element* min'. *Define the smallest element* $min \in T \wedge min' = f\ min$. *min is indeed the smallest element, because if there is a smaller element b' in T,* $b' < f\ min$ *holds. This would mean, by definition of T from S that there is a f b smaller than f min. However, this directly contradicts the well foundedness of S.*

- To show that the lexicographical ordering is well founded, we first define it as:

  Definition 3.4 $\forall a, c \in A\ b, d \in B.R(a, b)(c, d) \equiv a < c \vee a = c \wedge b < d$

  We then have to prove that each subset has a smallest element.

  Proof 3.5 *We can construct the smallest element directly. By taking the smallest element m from set A, and then look at all pairs (m,n) and take the smallest n from that set.*

Now that the well foundedness of the resulting function is established we can use it to derive induction schemes by means of the following theorem.

Theorem 3.6 $WF(<) \rightarrow \forall P.(\exists x.P(x) \rightarrow \forall y.(x < y \rightarrow P(x) \rightarrow P(y)) \rightarrow \forall x.P(x))$

The instantiation of this theorem with the derived ordering is available to the user. Whenever it is applied to a recursive function call within the body of a function definition it raises for the user the obligation to prove that in this particular instance the arguments arranged according to the derived induction scheme are smaller than when entering the function definition. The program then deduces that the property holds for that recursive function call.

If the well founded ordering creates subgoals that can not be proven, the algorithm permits the user to supply measure functions for all the different possible function calls. These measure functions must map the arguments of a function call onto a natural number or string. From then on, the algorithm uses an induction scheme based on the ordering provided by the $<$ operator for integers and strings.

3.2 Example: tactic applied to gcd

For our earlier defined gcd function we want to prove that the greatest common divisor of all even numbers is bigger than 1.

Property 3.7 $\forall x\ y.\,\gcd\ (x+x)\ (y+y) > 1$

When we apply this induction tactic to this proposition, specifying that we want it to derive an induction scheme which follows the recursive structure of the gcd function, the algorithm will determine which arguments are strongly connected. It will find that both arguments are, so the ordering imposed on them will be $R((x1,y1),(x2,y2)) = x1+y1 < x2+y2$. This ordering, combined with the well-foundedness theorem will yield this induction scheme:

Proposition 3.8 $\forall P.(\forall x1\ x2\ y1\ y2.x1+y1 < x2+y2 \rightarrow (P(x1,y1) \rightarrow P(x2,y2))) \rightarrow \forall xy.P(x,y)$

The proof then proceeds by reducing the gcd function and splitting it into each of the different cases. The user has to prove that the property holds when one of the arguments is $\perp$ or 0. No induction hypothesis is needed for them.

Only when the user has to prove that the property holds for either $\gcd\ (x-y)\ y$ or $\gcd\ x\ (y-x)$ an induction hypothesis is needed due to the recursive function call. Both proofs are nearly identical, so we assume we just have to prove:

Goal 3.9 $\gcd\ ((x+x)-(y+y))\ (y+y) > 1$

At that point an instantiation of our induction scheme will show that if we can prove that $x-y+y < x+y$, the property holds for the recursive function call.

Proof 3.10 *This is easy, since the case* $y = 0$ *has already been handled, we may assume that* $\gcd((x-y)+(x-y))\ (y+y) > 1$*, which is the same as* $\gcd((x+x)-(y+y))\ (y+y) > 1$.

3.3 Related work

Much research has been done the past few years to find the proper induction schemes that will make proving properties easy. Deepak and Shakur have used the coverset induction principle [D.K96]. Boulton and Slind use a

multi predicate induction scheme and a proof procedure to find the proper induction hypotheses to use [Bou00].

Tesketh uses middle out reasoning to guide the matching of induction hypotheses [Hes91] and Slind describes a method to derive induction schemes from translating function definitions into higher order logic [Sli97].

A few more authors have done closely related work by proving termination by means dependency graphs or call graphs. The earliest work was done by Arts and Giesl in, [TA97]. They translate functional programs into term rewrite systems and use dependency pairs to construct a well founded ordering. Abel and Altenkirch have written a system called foetus [AA02]. It only works for structurally smaller arguments however. Other research was done by Lee, Jones and Ben-Amram, described in [LJBA01]. They construct call graphs and prove termination by showing that if the program does not terminate it will contain an infinitely long chain of elements taken from a well founded set.

All those methods have in common that they require that the program analyzing the graph only considers arguments that are either known to get smaller or remain the same. Although this guarantees that termination is proven, sometimes it can prove to be more useful to leave the termination proof for later and to "guess" a well founded ordering.

Our tactic differs from the other methods in the way a well founded relationship is built using the information of the strongly connected components of the complete call graph.

3.4 Our method compared to other methods

The algorithm we have constructed creates a well founded relationship, however, it usually does not automatically prove that all recursive function calls have smaller arguments according to this relationship. Theoretically this may be considered to be a weaker result than most other methods that only look at arguments that they know are either equal or smaller. However, this weaker result is more powerful in practice as is shown below. Postponing the proof that the well founded ordering is a termination relationship has an advantage: The rest of the machinery of the theorem prover can be used to deduce that indeed the arguments are getting smaller.

- For instance in the following program on natural numbers, the information that the arguments are getting smaller is contained within the guards of the program. A program that simply looks at the arguments will not be able to deduce that the first argument is indeed getting smaller.

```
f x y z
| y < z = f (x + y - z) y z
| z < y = f (x - y + z) y z
= f (x-1) y z
```

- Another instance where programs that look at arguments will fail is if the information is hidden within the predicate that one is trying to prove. Take the following function and predicate:

  ```
  f x y
  | x = 0    = y
  = f (x-y) y
  ```

  $\forall x\ y.y > 0 \rightarrow$ `f` $x\ y = y$.

  The only way the tactic can conclude that the argument is getting smaller each consecutive call is if it knows the precondition that can be derived from the predicate.

- A third instance where the methods from the related work section fail is in programs where one argument is increasing, but not as hard as the argument it is combined with is decreasing.

  ```
  f x y    = g (x+1) (y-2) g
  g x y    = h x+y-1
  h x
   | odd x = f x/2 x/2
   = f ((x/2)-1) x/2
  ```

  None of the tactics that look at arguments that are getting smaller or stay the same can derive a well founded ordering for this set of functions, because the argument of h in the definition of g is not smaller than x or y.

All the methods mentioned in related work will not be able to derive an induction scheme for the above two mentioned cases. Another added advantage of guessing which variables are important and constructing a well founded relationship out of them is that unlike for instance the size change principle from Lee, Jones and Ben-Amram [LJBA01], the algorithm is not PSPACE hard. Tarjan's algorithm's complexity is in the order of the number of the edges and vertices combined.

It must also be mentioned that even if termination is proven, it is not immediately clear what kind of induction scheme can be derived from the

termination proof. After all, an induction hypothesis may only be invoked if an element of a call sequence is smaller and the algorithm of Lee, Jones and Ben-Amram only deduces it must decrease at some point, but not at which point.

Of course the downside of guessing a well founded relationship is that one can be wrong. The algorithm we construct will fail if the algorithms that need to be analyzed start with arguments that increase. A way to address this problem is by letting the programmer indicate which variables can be ignored in the derivation of the well founded ordering.

To see how our approximation algorithm works compared to the earlier mentioned procedure from Lee, Jones and Ben-Amram, we have taken the examples from their article and checked how we can prove them in Sparkle using our new tactic.

- Reverse function with accumulating parameter.

  ```
  rev ls = r1 ls []
  r1 [] a = a
  r1 [l:ls] a = r1 (ls) [l:a]
  ```

  For this function will be deduced that the first and the second parameter of r1 are in different strongly connected components. Therefore the ordering will be the lexicographical product of both arguments. The subsequent proof obligation for the recursive function call of r1 will be that $\texttt{size ls} < \texttt{size } [\texttt{l} : \texttt{ls}] \vee (\texttt{size ls} == \texttt{size } [\texttt{l} : \texttt{ls}] \wedge \texttt{size } [\texttt{l} : \texttt{a}] < \texttt{size a})$.

  This is trivial to prove.

- Program with indirect recursion.

  ```
  f []     x = x
  f [l:ls] x = g ls x l
  g a b c = f a [c:b]
  ```

  For this function the algorithm will detect that (g a) and (f ls) are in the same strongly connected component and (g b), (g c) and (f x) are in the same strongly connected component. That means that the proof obligation of the call to g from function definition f will be that $\texttt{size ls} < \texttt{size } [\texttt{l} : \texttt{ls}] \vee (\texttt{size x} + \texttt{size } [\texttt{l} : \texttt{ls}] < \texttt{size x})$

  While the proof obligation for the recursive function call of f from g will be $\texttt{size a} < \texttt{size a} \vee \texttt{size b} + \texttt{size c} < \texttt{size b} + \texttt{size c}$. This latter

condition will never be true, so it is impossible to use the induction hypothesis.

The solution is to reduce g in the proof of the property of f. Then one can use the f to f relationship, which states that `size ls` $<$ `size` $[$`l : ls`$]\vee$

(`size ls` $=$ `size ls` $\wedge$ `size x` $<$ `size x`). The proof of this is straightforward.

- Function with lexically ordered parameters.

```
a m n
| m == 0 = n + 1
| n == 0 = a (m-1) 1
= a (m-1) (a m (n-1))
```

For this program we will find that m and n are both in different components, so the algorithm will construct the lexicographical ordering on m and n to create an induction scheme. This means that the proof obligation for the first call of a is $m-1 < m \vee (m-1 == m \wedge n <$ `a` $m\ (n-1))$ and for the second call $m < m \vee m == m \wedge n - 1 < n$. All trivially easy to prove.

- Program with permuted parameters.

```
p m n r
| r > 0 = p m (r-1) n
| n > 0 = p r (n-1) m
= m
```

This program has all arguments in the same strongly connected component. The proof obligation in order to use the induction hypothesis for the first call to p will therefore be $m+n+r-1 < m+n+r$, while for the second it will be $r+n-1+m < m+n+r$. Both are again easily proven.

- Program with permuted and possibly discarded parameters.

```
f x [] = x
f [] y = f y (tl y)
f x y  = f y (tl x)
```

Again both arguments are in the same component. This means that for the first recursive call of f `size y` $+$ `size` `(tl y)` $<$ `size y` should

hold. Unfortunately it does not, so the program will have guessed the wrong well founded relationship. It is up to the user to provide a measure function. For instance: `measure x y = size y`. By supplying the algorithm with this function, the user has to prove for the first function call that `size (tl y)` $<$ `size y` and for the second function call a further reduction of f and a split case tactic on the variable x is needed to find that `size (tl y)` $<$ `size y`, before the induction hypothesis may be used.

- Program with late-starting sequence of descending parameter values.

```
f a []      = g a []
f a [b:bs]  = f [b:a] bs
g [] d      = d
g [c:cs] d  = g cs [c:d]
```

The algorithm will conclude that all the elements are in different strongly connected components. That means that a lexical ordering is created for all the arguments. The following proposition does not hold:

`size [b : a]` $<$ `size a` $\vee$ (`size [b : a]` $==$ `size a` $\wedge$
`size [b : bs]` $<$ `size [bs]`)

However, properties of a function like this can much more easily be proven by induction on the second argument of f and the first argument of g separately.

In the previous sections we have shown how a complete call graph is built from the function definitions and how a well founded relationship is constructed by means of the strongly connected components of that graph. This relationship is the basis of an induction scheme. We have shown for what kind of programs our method will yield an induction scheme where other methods fail. And by means of examples, taken from another research paper, we have shown that in practice, the examples which are intended to show the advantages of a theoretically strong method like the size change principle, can easily be proven using Sparkle extended with our new tactic.

4 Acknowledgements

I would like to thank all the participants of the Fifth Symposium on Trends in Functional Programming for their valuable input.

5 Future work

R. van Kesteren is enhancing Sparkle with proof rules for parameterized classes [van04]. This will make proofs of properties of generic classes possible.

The authors are also working on a new co-inductive technique, based on guard-ed co-induction without the syntactic constraint that is required in other methods.

Furthermore, inclusion of features of Clean like dynamics and overloading, for which there is limited support now within Sparkle, deserve consideration.

6 Conclusion

By extending the induction and co-induction proof techniques for Sparkle, we have opened the theorem prover up for a wide set of proofs on properties of programs. We have devised a new method to derive induction schemes for mutually recursive functions, that closely match the structure of the underlying program. This method can create induction schemes to prove properties of functional programs where other methods fail. Combined with the already easy to use interface, a programmer can now easily prove properties of recursive and co-recursive programs, bringing the integration of programming and proving closer.

References

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[BJ96] J. Rutten B. Jacobs. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, pages 222–259, 1996.

[Bou00] Richard Boulton. Multi-predicate induction schemes for mutual recursion. Informatics research report EDI-INF-RR-0046, University of Edinburgh, 2000.

[D.K96] M. Subramaniam D.Kapur. Automating induction over mutually recursive functions. In *Proceedings of the 5th international conference on algebraic methodology and software technology*, vol-

ume 1101 of *Lecture notes in computer science*. Springer Verlag, 1996.

[Hes91] J.T. Hesketh. *Using Middle-Out reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1991.

[JG04] G. Hutton J. Gibbons. Proof methods for corecursive programs. *Fundamenta Informaticae XX*, pages 1–13, 2004.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.

[McI99] M. Douglas McIlroy. Functional pearls: Power series, power serious. *Journal of Functional Programming*, 9:323–335, 1999.

[MdM01] R. Plasmeijer M. de Mol, M. van Eekelen. Theorem proving for functional programmers. *LNCS:Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, 2001.

[MJCG93] T. F. Melham M. J. C. Gordon. *A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[MvE01] R. Plasmeijer M. van Eekelen. *Concurrent Clean Language Report (version 2.0)*. University of Nijmegen, 2001.

[Pau87] L.C. Paulson. *Logic and Computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[Sli97] Konrad Slind. Derivation and use of induction schemes in higher order logic. *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 275–290, 1997.

[TA97] J. Giesl T. Arts. Automatically proving termination where simplification orderings fail. In *Proceedings Colloquium on Trees in Algebra and Programming*, volume 1214, pages 261–272. Berlin: Springer-Verlag, 1997.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM journal of Computing*, pages 146–160, 1972.

[van04] van Kesteren, Ron, van Eekelen, Marko, and de Mol, Maarten. An Effective Proof Rule for General Type Classes. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 149–165. Ludwig-Maximilians Universität, München, November 2004.